Projet 1 du cours INFO-F-403

THÉORIE DES LANGAGES ET DE LA COMPILATION

PASCAL

Nicolas MAQUET

Année Académique 2009–2010

Comparing C and Pascal is rather like comparing a Learjet to a Piper Cub - one is meant for getting something done while the other is meant for learning.

Brian W. Kernighan

1 Introduction

Le Pascal est un langage de programmation procédural développé vers la fin des années soixante dont le but principal a été l'apprentissage de la *programmation structurée* aux étudiants. Comme son nom l'indique, la programmation structurée consiste à organiser le code source en de multiples sous-routines indépendantes, aussi appelées *fonctions* ou *procédures*. Une des vocations principales du Pascal était donc d'apprendre à programmer "sans `goto`", bien que pour des raisons mystérieuses la maudite instruction fasse effectivement partie du language.

Au delà de son usage purement académique, le Pascal a évolué en de nombreux dialectes, dont certains ont eu du succès dans l'industrie comme *Object Pascal* ou *Borland Delphi*. Aujourd'hui, le Pascal n'est plus considéré comme un langage de programmation de premier plan, mais il laisse derrière lui un vénérable héritage que vous allez pouvoir honorer à votre tour en lui dédiant un compilateur écrit par vos soins !

2 Le Pascal

Le langage Pascal insiste très fort sur les déclarations ; toute variable, procédure ou fonction doit impérativement être déclarée *avant* de pouvoir être utilisée. Ceci a notamment pour but de rendre la compilation plus simple ; comme la nature de chaque identifiant est connue lorsque le compilateur la rencontre, le contrôle de type et la génération de code peuvent se faire *en parallèle* [1]. Une autre particularité du Pascal est que le point virgule est un *séparateur* d'instruction et non un *terminateur* d'instruction comme en C ou en Java. Contrairement au C où toutes les routines sont appelées *fonctions*, le Pascal distingue les routines qui ne renvoient pas de valeur (appelées procédures) de celles qui en renvoient (appelées fonctions). En Pascal, les commentaires sont entourés d'accolades. Les identifiants du Pascal (pour les variables, procédures, fonctions et le nom du programme) doivent appartenir au langage de l'expression régulière `[a-zA-Z]([a-zA-Z]|[0-9]|_)*`.

1. On parle généralement de compilation *en une passe*.

Dans le cadre de ce projet, vous devrez réaliser un compilateur pour une version largement simplifiée du Pascal. Tout d'abord, parmi les nombreux types de données différents du langage votre compilateur ne devra en supporter que deux : `integer` et `boolean`. Aussi, votre compilateur n'aura pas à gérer les programmes qui gèrent leur mémoire de manière dynamique, ce qui se fait normalement en Pascal par le biais des primitives `new` et `dispose` (équivalents des `new` et `delete` du C++). Enfin, le Pascal supporte normalement la déclaration de fonctions et procédures *imbriquées*, ce que votre compilateur devra interdire. Ce dernier devra néanmoins être capable de compiler des programmes contenant des fonctions et procédures traditionnelles.

Il n'est pas possible en Pascal de donner une *valeur initiale* aux variables déclarées ; votre compilateur devra initialiser les entiers à zéro et les booléens à *false*. La figure 2 donne un exemple de programme en Pascal simplifié. Vous remarquerez que la manière dont on spécifie dans le code qu'une fonction doit retourner une valeur est assez étrange ; en effet, il faut *assigner*[2] à l'identifiant de la fonction la valeur souhaitée. Ceci peut être fait plusieurs fois, et même aucune fois (auquel cas la fonction renverra zéro si elle est entière, et *false* si elle est booléenne).

3 Machine virtuelle

Le but d'un compilateur étant tout de même la génération de code, il est temps d'introduire la "machine" vers le langage de laquelle nous allons compiler les programmes. Il s'agit de la GPMACHINE, qui fonctionne avec une pile et qui interprète des instructions en PCode. Vous trouverez cette machine virtuelle à l'adresse : `http://www.info.fundp.ac.be/~gpm`. Un mode d'emploi pour GPMACHINE est disponible sur la page web du TP.

Quelques unes des instructions de la GPMACHINE dont vous aurez besoin n'ont pas été vues au TP. Le tableau de la figure 4 reprend ces instructions additionelles. Ces instructions utilisent la notion de *stack frame*, c'est à dire une portion du stack qui est réservée soit à l'exécution du bloc principal du programme (*nesting* 0) soit à l'exécution d'une procédure ou fonction (*nesting* 1). Comme mentionné précédemment, le Pascal permet normalement la déclaration de fonctions et procédures imbriquées (de *nesting* supérieur à 1) mais vous ne devez pas vous en soucier. Pour appeler une fonction ou procédure, vous devez procéder comme suit :

- `mst N` : alloue une nouvelle stack frame (`N` est le nesting courant)
- mettre les paramètres de la procédure ou fonction au sommet de la pile
- `cup P L` : appelle la fonction ou procédure d'étiquette `L` avec `P` paramètres.
- au début de la fonction ou procédure appelée, les paramètres sont au sommet de la pile
- exécution du code de la fonction ou procédure
- `retf` (pour une fonction) ou `retp` (pour une procédure) pour effectuer le retour à l'appelant
- si l'appelé était une fonction, la valeur de retour est au sommet de la pile

L'instruction `mst N` (*mark stack*) crée une nouvelle stack frame où `N` est le *nesting* courant, et a pour effet d'allouer 5 nouvelles cases sur le stack. Vous pouvez traiter les 4 cases du dessus comme une "boîte noire" ; son contenu sert aux instructions `retf`, `retp`, `lod`, `cup` et `str` mais vous ne devez pas les utiliser directement. Par contre, la première des 5 cases mise sur le stack par `mst` sert à communiquer la valeur de retour d'un appel de fonction et vous serez donc amenés à modifier son contenu.

Les instructions `lod N A` et `str N A` permettent d'accéder aux cases mémoire du stack ; le

2. Le Pascal dispose normalement aussi d'une instruction *return* classique, qu'on ne vous demande pas d'implémenter.

paramètre `N` doit valoir 0 si on veut accéder aux cases du bloc courant et 1 si on veut accéder aux cases du bloc de nesting inférieur ; le paramètre `A` est un *décalage* par rapport au début de la *stack frame* courante. Pour prendre un exemple concret, si on exécute le programme de la figure 3 jusqu'au deuxième passage à la ligne 4 (on suppose que l'utilisateur veut calculer la factorielle de 5) on obtient le contenu du stack décrit à la figure 5. Pour accéder à la variable globale `b` le programme utilise les instructions `lod 1 1` et `str 1 1`. Pour accéder au paramètre `x` qui se trouve au sommet de la pile, le programme utilise l' instruction `lod 0 5` puisqu'elle se trouve à un décalage 5 du début de la stack frame courante. Enfin, pour accéder à la valeur de retour de la fonction (la *pseudovariable* `Euclide`) le programme utilise l'instruction `str 0 0` puisque la valeur de retour est toujours la première valeur de la stack frame. Vous remarquerez qu'il n'est pas possible d'accéder aux stack frames situées entre la stack frame courante et la stack frame globale ; ceci correspond au comportement attendu : une procédure ou fonction peut accéder au variables globales et à ses variables locales, mais pas aux variables locales des appels intermédiaires.

Pour vous familiariser avec ces instructions, nous vous conseillons de lire attentivement le code de la figure 3 et de l'exécuter avec la GPMACHINE (le code de la figure 3 est disponible en version électronique sur la page web de l'assistant : `http://www.ulb.ac.be/di/ssd/nmaquet/`).

4 *Desiderata*

Étant donnée la grammaire du Pascal simplifié, nous vous demandons d'implémenter un compilateur pour ce langage. Vous pouvez choisir de l'implémenter en C ou en C++ [3]. Plus précisément nous attendons de votre part le travail suivant :

1. Identifiez les *unités lexicales* du Pascal simplifié ;
2. Donnez un automate fini déterministe les reconnaissant ;
3. Écrivez un analyseur lexical, qui reconnaît les différentes unités lexicales du Pascal simplifié. Celui-ci sera implémenté sous forme d'une fonction qui, à chaque appel, retournera la prochaine unité lexicale lue sur l'*input*. Une table des symboles sera également implémentée ;
4. Modifiez la grammaire du Pascal simplifié de telle sorte que :
   (a) Celle-ci soit LL(1) ;
   (b) Elle tienne compte de la *priorité* et de l'*associativité* des opérateurs, à savoir (du plus au moins prioritaire) :

Opérateur(s)	Associativité
`not`, `-`	à droite
`*`, `/`, `and`	à gauche
`+`, `-`, `or`	à gauche
`>`, `<`, `>=`, `<=`, `<>`, `=`	à gauche

5. Donnez la table de *parsing* LL(1) pour cette grammaire, ainsi que les éventuels calculs de First() et de Follow() qui vous auront été nécessaires ;
6. Décorez la grammaire en lui ajoutant les instructions nécessaires pour (i) effectuer le contrôle de type, et (ii) produire du code pour la machine virtuelle ;
7. Écrivez un analyseur LL(1) en descente récursive, produisant du code pour la machine virtuelle dans un fichier texte.

3. Si vous désirez l'implémenter dans un autre langage, consultez d'abord l'assistant.

8. Votre compilateur devra effectuer un *contrôle de type* systématique du code. Chaque expression peut-être soit booléenne soit entière et le compilateur devra renvoyer une erreur le cas échéant (par exemple `a < 3 and 2` est erroné). De plus, les appels de fonctions et procédures doivent êtres vérifiés pour s'assurer que le nombre de paramètres et leurs types soient corrects.
9. Nous ne vous demandons pas de gérer le *hiding* des variables globales. Si une variable locale à une fonction ou procédure a le même nom qu'une variable globale, votre compilateur peut renvoyer un erreur. Dans le cas contraire, les variables globales devront bien sûr être accessibles depuis l'entièreté du programme, comme dans l'exemple de la figure 3.

Le tout sera naturellement présenté dans un *rapport*, qui reprend les différentes informations demandées, et qui précise et justifie les choix et hypothèses que vous aurez faits. Le code source de votre compilateur (accompagné de ses *Makefile*, de ses fichiers de test et de tout autre fichier nécessaire pour se convaincre que le travail a été bien fait) doit être envoyé par e-mail à `nmaquet@ulb.ac.be`. Le tout doit être remis pour le vendredi 26 février 2010. Le travail peut être fait par groupe de deux (ce que nous recommandons), ou individuellement. Une **défense orale** sera programmée lors de la correction du projet.

Bon travail !

[1] <PROGRAM> → program ID ; <VARS> <ROUTINELIST> <BLOCK> . $
[2] <VARS> → var <IDLIST> : <TYPE> ; <VARS>
[3] → ε
[4] <IDLIST> → ID , <IDLIST>
[5] → ID
[6] <TYPE> → integer
[7] → boolean
[8] <ROUTINELIST> → <ROUTINE> ; <ROUTINELIST>
[9] → ε
[10] <ROUTINE> → <FUNCTION>
[11] → <PROCEDURE>
[12] <FUNCTION> → function ID <ARGSOPT> : <TYPE> ; <VARS> <BLOCK>
[13] <PROCEDURE> → procedure ID <ARGSOPT> ; <VARS> <BLOCK>
[14] <ARGSOPT> → (<ARGS>)
[15] → ε
[16] <ARGS> → <IDLIST> : <TYPE> ; <ARGS>
[17] → <IDLIST> : <TYPE>
[18] <BLOCK> → begin <STATEMENTOPT> end
[19] <STATEMENTOPT > → <STATEMENTLIST>
[20] → ε
[21] <STATEMENTLIST > → <STATEMENT>
[22] → <STATEMENT> ; <STATEMENTLIST>
[23] <STATEMENT> → ID := <EXPRESSION>
[24] → <CALL>
[25] → <BLOCK>
[26] → if <EXPRESSION> then <STATEMENT> else <STATEMENT>
[27] → while <EXPRESSION> do <STATEMENT>
[28] → read (ID)
[29] → write (<EXPRESSION>)
[30] <CALL> → ID
[31] → ID (<EXPRESSION>)
[32] <EXPRESSIONLIST> → <EXPRESSION>
[33] → <EXPRESSION> , <EXPRESSIONLIST>
[34] <EXPRESSION> → ID
[35] → NUM
[36] → true
[37] → false
[38] → (<EXPRESSION>)
[39] → - <EXPRESSION>
[40] → not <EXPRESSION>
[41] → <EXPRESSION> <OP> <EXPRESSION>
[42] <OP> → <>
[43] → <
[44] → >
[45] → <=
[46] → >=
[47] → =
[48] → +
[49] → -
[50] → *
[51] → /
[52] → and
[53] → or

FIGURE 1 – La grammaire du Pascal simplifié

```
program Example;

var x, y: integer;

function Euclide (var a, b : integer) : integer;
var c : integer;
begin
    while b <> 0 do
    begin
        c := b;
        while a >= b do
        begin
            a := a - b  { Calcul de a modulo b }
        end;
        b := a;
        a := c
    end;
    Euclide := a
end;

procedure WriteUnary(var a: integer);
begin
    while a > 0 do
    begin
        write(1);
        a := a - 1
    end
end;

begin
    read(x);
    read(y);
    x := Euclide(x, y);
    write(x);
    WriteUnary(x)
end.
```

FIGURE 2 – Un exemple de programme en Pascal simplifié.

```
ldc i 0            | 01 | program Factorial;
ldc i 0            | 02 | var a, b : integer;
ujp @main          | 03 |
define fact        | 04 | function fact (var x : integer) : integer;
lod i 1 1          | 05 | begin
ldc i 1            | 06 |    b := b + 1;
add i              | 07 |
str i 1 1          | 08 |
lod i 0 5          | 09 |    if x = 0 then
ldc i 0            | 10 |
equ i              | 11 |
fjp fact1          | 12 |
ldc i 1            | 13 |         fact := 1
str i 0 0          | 14 |
ujp fact2          | 15 |
define fact1       | 16 |    else
mst 1              | 17 |         fact := x * fact(x-1)
lod i 0 5          | 18 |
ldc i 1            | 19 |
sub i              | 20 |
cup 1 fact         | 21 |
lod i 0 5          | 22 |
mul i              | 23 |
str i 0 0          | 24 |
define fact2       | 26 |
retf               | 27 | end;
define @main       | 28 | begin
read i             | 29 |     read(a);
str i 0 0          | 30 |
ldc i 0            | 31 |     b := 0;
str i 0 1          | 32 |
mst 0              | 33 |     a := fact(a);
lod i 0 0          | 34 |
cup 1 fact         | 35 |
str i 0 0          | 36 |
lod i 0 0          | 37 |     write(a);
prin               | 38 |
lod i 0 1          | 39 |     write(b)
prin               | 40 |
stp                | 41 | end.
```

FIGURE 3 – Un exemple de programme en PCode et son équivalent Pascal.

`lod T N A`	Met (push) sur la pile une copie de la valeur (de type `T`) stockée à l'adresse (`N`, `A`).
`str T N A`	Déplace (pop) le sommet de la pile (de type `T`) vers l'adresse (`N`, `A`).
`mst N`	Crée une *stack frame* en vue d'un appel de fonction ou procédure (*nesting* courant = `N`).
`cup P L`	Appelle la fonction ou procédure à l'étiquette `L` avec `P` paramètres.
`retp`	Effectue un retour d'appel de procédure.
`retf`	Effectue un retour d'appel de fonction.

FIGURE 4 – Instructions de la GPMACHINE non vues au TP.

adresse absolue	contenu du stack	variable	adresse `N A`
13	4	x	0 5
12	-	-	0 4
11	-	-	0 3
10	-	-	0 2
9	-	-	0 1
8	non défini	Euclide	0 0
7	5	x	-
6	-	-	-
5	-	-	-
4	-	-	-
3	-	-	-
2	non défini	Euclide	-
1	1	b	1 1
0	5	a	1 0

FIGURE 5 – Contenu du stack à l'éxecution du programme de la figure 3 (2^e appel à *fact*, ligne 4).